File System Implementation

Disk Structure

- **Partitions**: disk can be subdivided into partitions
- **Raw usage**: disks/partitions can be used raw (unformatted) or formatted with file system
- **Volume**: entry containing FS
  - tracks that file system's info is in device directory or volume table of contents
- **FS diversity**: there are general purpose and special purpose FS

File Systems — Logical vs. Physical

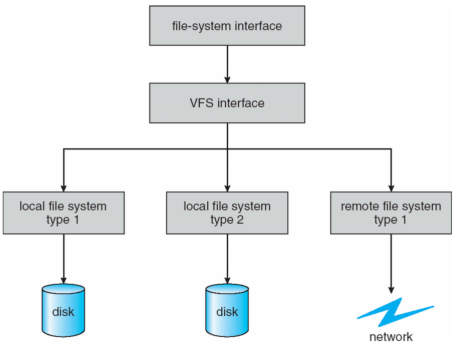
- **Logical**: can consist of different physical file systems
- **Placement**: file system can be mounted at any place within another file system
- **Mounted local root**: bit in i-node of local root in mounted file system identifies this directory as mount point

File Systems — Layers

- **Layer 5**: applications
- **Layer 4**: logical file system
- **Layer 3**: file-organization module
- **Layer 2**: basic file system
- **Layer 1**: I/O control
- **Layer 0**: devices

File Systems — Virtual

- **Principle**: provide object-oriented way of implementing file systems
  - same API used for different file system types



Files — Implementation

- **Meta data** must be tracked:
  - which logical block belongs to which file?
  - block order?
  - which blocks are free for next allocation?
- **Block identification**: blocks on disk must be identified by FS (given logical region of file)

  → meta data needed in *file allocation table*, *directory* and *inode*
- **Block management**: creating/updating files might imply allocating new/modifying old disk blocks

Allocation — Policies

- **Preallocation**:
  - *problem*: need to know maximum file size at creation time
  - often difficult to reliably estimate maximum file size
  - users tend to overestimate file size to avoid running out of space
- **Dynamic allocation**: allocate in pieces as needed

Allocation — Fragment size

- **Extremes**:
  - fragment size = length of file
  - fragment size = smallest disk block size (= sector size)
- **Trade-offs**:
  - *contiguity*: speedup for sequential accesses
  - *small fragments*: larger tables needed to manage free storage and file access
  - *large fragments*: improve data transfer
  - *fixed-size fragments*: simplifies space reallocation
  - *variable-size fragments*: minimizes internal fragmentation, can lead to external fragmentation

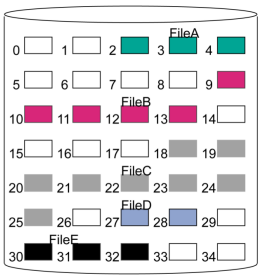
Allocation — File space

- **Contiguous**
- **Chained**
- **Indexed**:
  - fixed block fragments
  - variable block fragments

characteristic	contiguous	chained	indexed	
preallocation?	necessary	possible	possible	
fixed or variable size fragment?	variable	fixed	fixed	variable
fragment size	large	small	small	medium
allocation frequency	once	low to high	high	low
time to allocate	medium	long	short	medium
file allocation table size	one entry	one entry	large	medium

Allocation — Contiguous

- **Principle**: array of n contiguous logical blocks reserved per file (to be created)
- **Periodic compaction**: overcome external fragmentation



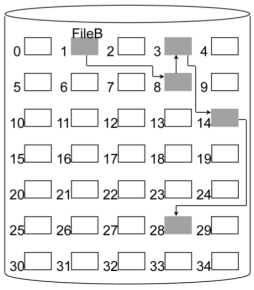
File Allocation Table

File Name	Start Block	Length
FileA	2	3
FileB	9	5
FileC	18	8
FileD	27	2
FileE	30	3

Allocation — Chained

- **Principle**: linked list of logical blocks per file
  - FAT or directory contains address of first file block

  → *no external fragmentation*: any free block can be added to chain



File Allocation Table

File Name	Start Block	Length
...	...	...
FileB	1	5
...	...	...

Allocation — Indexed

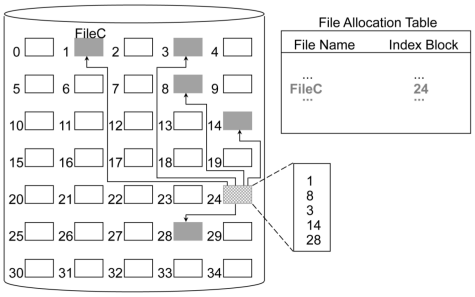
- **Principle**: FAT contains one-level index table per file
  - *generalization*: n-level index table
  - index has one entry for allocated file block
  - FAT contains block number for index

Directories — Implementation

- **Simple directory** (MS-DOS):
  - fixed-size entries
  - disk addresses + attributes in directory entry
- **i-node reference directory** (UNIX):
  - entry refers to i-node containing attributes

Disk Blocks — Buffering

- **Buffering**: disk blocks buffered in main memory
- **Access**: buffer access done via hash table
  - blocks with same hash value are chained together
- **Replacement**: LRU
- **Management**: free buffer is managed via doubly-linked list



File Systems — Journaling

- **Principle**: record each update to file system as *transaction*
  - written to log
- **Committed** transaction = written to log
  → *problem*: file system may not yet be updated
- **Writing** transactions from log to FS is asynchronous
- **Modifying** FS → transaction removed from log
- **Crash** of file system → remaining transactions in log must still be performed

File Systems — Log-structured

- **Principle**: use disk as circular buffer
  - write all updated (including i-nodes, meta data and data) to end of log
- **Buffering**: all writes initially buffered in memory
- **Writing**: periodically write within 1 segment (1 MB)
- **Opening**: locate i-node, find blocks
- **Clearing**: clear all data from other end, no longer used